From 386BSD to OSPREY
The Evolution of an Operating System

Lynne Greer Jolitz[1]
lynne@telemuse.com
Founding Partner
TeleMuse

Abstract
While 386BSD is famous for pioneering inexpensive 386-based Unix systems for academics and open source code for programmers, it also proved to be a fungible framework for new approaches, unfettered by legacy or short-term commercial objective. This paper examines a single aspect of OS operation in the form of tracking root resource usage in a server, specifically processor and memory. This aspect is extremely relevant to all corporate datacenters, as the cost effectiveness of banks of as many as hundreds of servers is directly related to how effectively these two are utilized. Poor utilization of resources means a datacenter needs more machines to do the same work, while perfect resource utilization would match resource to need exactly. The author believes that the findings generalize to most other aspects of servers, as they derive efficiencies from the underlying effectiveness of these root resource – issues 386BSD addressed years ago. Due to the structure of Unix itself, however, some root resource legacies can only be conquered with a radical new approach, inspiring the OSPREY system.

Introduction
The 386BSD open source operating system was the first fully documented open source port of Berkeley UNIX to the 386 architecture. Finally released in 1992 after 2 years of comprehensive articles and source code in *Dr. Dobbs Journal* and mirrored throughout the Internet, over 250,000 downloads were obtained by users in a single week – a harbinger of the Internet as *communi consensu*. Follow-on releases, enhanced by contributions all over the world, followed over the next two years.

The 386 platform was chosen to provide a widespread and economic base for operating systems development and design. 386BSD primary beneficiaries were smaller universities, research groups, and individuals on four continents who could not afford to run BSD on the then more expensive computer systems from IBM, HP, and DEC. Even Tiny 386BSD, a single-floppy complete Berkeley UNIX system, was used to encourage small contributions to charity under the magazine's auspices to benefit women and children.

But 386BSD design philosophies were also relevant – and are perhaps even more relevant now than a decade ago. Operating systems can no longer be arbitrary and utilitarian, but instead intensely pragmatic in design and architecture. The success of the original UNIX system, carried on in generations of Berkeley Unix, came from the devotion to minimalism. Plan 9 also echoed this devotion in its file-centric design. But in the distraction of conversion and religious wars, Linux, BSD, and even Windows now suffer from terminal design flaws which tenaciously linger as they migrate throughout the software. As mounds of code are added, the fragility of the system itself increases, and frustration of both designers and consumers grows as they face intractable problems, such as servers in the datacenter that make poor use of resource and difficult-to-isolate resource bottlenecks (e.g. processor, memory and abstract resources).

The Internet datacenter (IDC) is an environment where heterogeneous elements are used and stressed to provide content to an end-customer. It is an environment where any fragility in an element like an operating system becomes a bottleneck to successful 24/7 operation. System administration costs incurred in the datacenter are often unpredictable, and due to these key design decisions, which for the above named platforms are essentially unalterable at this advanced date. The uniqueness of a research system like 386BSD as pioneered by those riotous academic days is that one need not be wedded to ideology and politics, and that work can proceed at its own pace. Many of these flaws had thus been examined with innovations pioneered in the Release 1 386BSD, and even further in OSPREY (Operating System for Research, Education, and Technology), a radical new approach. These two projects, built upon each other over the last decade, provide a "present" and "future" roadmap in operating systems design required for the next decade.

Processors and Memory in the Internet Data Center
In the IDC, it is often presumed that a "slow" server implies a slow processor. The underlying operating system is considered too arcane to understand – it is something that is upgraded upon demand. The server as hardware is straightforward, consisting of a processor (or set of processors), RAM, disk storage, operating system, utilities, and applications. But it is how the server is used which truly differentiates it, and no longer as much the hardware it contains.

Processors are the server's engines, generating and transforming information. Through a variety of tools we watch the server's processor performance in action, just as a racing car's tachometer makes visible to the driver the best times to shift between different gears to win the race. Our choice of tools determines how to apply this information to remedy problems – these tools shine a light on poor processor performance, processor configuration, and processor reliability. We may wish to know, for example, how much of a processor is being consumed on each server. From this information alone, the entire cage may be torn apart as a server is reconfigured, or a cookie-based load balancer added.

[1] Lynne Greer Jolitz, co-creator of 386BSD and OSPREY, is the author of many articles and books on operating systems and networking design. She is the creator of SiliconTCP, a dataflow ballistic protocol mechanism. Her most recent book is on Internet datacenter (IDC) technology and operations. She received her degree in physics from the University of California at Berkeley.

Memory is the gatekeeper of server capacity, mediating how well the processor and the server may be used. Adding memory can improve a server's performance by alleviating the need for the operating system to buy space by paging in and out information to slower storage, thus allowing a processor to achieve a higher sustained rate of computation. Memory is an indirect way to measure performance, however, and adding DRAM ignorantly may not yield the desired boost in performance. Unlike processors, memory effects are pervasive through the fabric of the data center.

An IDC may require its servers work together to complete an operation. Resource can be partitioned through the use of load balancing equipment, or through application and content distribution. It may even be balanced through clustered processor arrangements, if the application has been suitably designed. However, whether a single processor or multiple processor arrangement is used, the same criteria of resource allocation and ability to bear prevails.

A simple examination of the processor queue is often misleading. A bottleneck or interdependency with some other process may result in the "appearance" of a slow processor. This interdependency is masked by the simple manner in which queues are examined, as either the processor queue or "everything else". Yet when more resource is added, the performance is still "slow". In fact, adding resource can result in additional delays or latency which may even degrade performance unexpectedly. The addition of memory resource under the simple assumption that the application is memory-bound may also return less-than-stellar results for the costs invested. Blind assumptions of simple processor or memory starvation often shoot in the dark at presumed problems.

We're not talking of deep, mystical analysis here. Most server management by system administrators boil down to "four function calculator" issues – sums of resources, rates of use, and subtractions to yield what the wasted (or inaccessible) resource amount is present. Calibration determines actual capacity and consumption patterns. Speeds and feeds couple with realizable duty cycles. IDC's run on pragmatic metrics.

To lessen consumptive process behavior, the operating system lowers the process scheduling priority (decays) as it consumes more processor resource, and regains priority when they become less active as a result – thus, other processes late to the resource still get their chance. The more processes consuming the processor resource, the faster the decay of priority. We might experience the situation where ample resources are available, yet not enough processor cycles are present to maintain the demand a server is under. In this case, the server is truly processor bound. In well-funded IDC, this is the uncommon case.

Consumption of Resource: Operating System Impact

Consumption of resource in the IDC is more than physical processors or memory – it is an intricate operating system issue. Resources in an operating system are viewed as an internal shared data abstraction that are either referenced, operated on, or reconciled (or all of these together) to achieve their desired result within the allocated time slice. As multiple resources are often needed to perform operating system functions for the application program, the assembling of the necessary set for an exclusive operation may take more than one time slice. Also, a program might not get all of the set at once and run out of time – it is then required to divest the partial set to prevent a server deadlock, and try again to assemble the set once the primary resource becomes free.

Since these resources may consume considerable memory, itself a resource underlying the mentioned data abstraction, contention for memory may alter the competition for which processes may be run first (e.g. those with the resources already owning memory). It may be so strained, that unless more memory is added to a system, the processor will never run at full capacity – this is a memory bound condition. Resources that turn over quickly may advance processes that depend on them, by being readily available when other more starved resources take longer to come available, through the use of good resource affinity scheduling.

Resources that are highly contended for by many processes, in contrast, may set a server capacity limit by their very existence. In other words, we may have tons of memory and processor, but with so much demand the weak point becomes the service allocation mechanism. This is called an *abstract resource bound* condition, and occurs frequently in socket allocation.

The simplest case of a processor bound condition is that of an infinite loop application. Multiple copies running effectively partition the processor's cycles into bounded limits, irrespective of the actual scheduling implementation and or preference. Thus, one could measure effectiveness of an operating system without relying on its own status or accounting mechnism, but instead by artificially degrading performance.

Most operating systems do an excellent job of managing processor and memory resources in the casual case. Where this begins to break down is where applications push the limits for good (relevant, actual load) or bad (poor use of resource, regardless of load) reasons. We begin the hunt to determine why by examining processor and memory usage.

Processor, Memory, and Abstract Resource Limits Implied by the Operating System

A server's processor can only execute a finite number of instructions per second to which an operating system can subdivide to process. If a process or group of processes absorb this capacity, we've the obvious case that we've exhausted the processor and become processor bound – we've got many requests with too little resource. We can be processor bound for many reasons: too much real work, too much wasted work, or too much fragmented work. Our first question is where is this work going? Server engineers love to use the process status command, ps on Unix systems, to sort on percent CPU (%cpu) utilization to examine at a gross level where the processor is being consumed.

For example, lets assume our Ghz processor has instructions that execute on the average of 10ns or 108 instructions/second. That would mean that if a time slice was 0.01 second, 1-1,000,000 instructions would be executed by a process in a time slice. Even a process taking a fraction of a percent of a processor may tie up fantastic numbers of instructions. Having the ps command sort the list of processes by percent cpu shows the current consumers. If a few processes consume a processor, they stand out on the listing with appreciable percentages you can direct your attention to. When hundreds to thousands of processes each have a fraction of a percent, ps is harder to use to point a finger at specific issues – you need to select classes of processes (like web server applications with a grep command), tally the percent cpu for all processes in the class (using the awk command), and then its possible to get the amount of processor resource dedicated to a function.

The IDC must determine if the processor spends the majority of its time in user mode or kernel mode. User mode is used to run applications code, while the operating system kernel runs in kernel mode at the behest of the operating system and its applications. Heavy load in kernel mode insures that little applications use can be made of the processor. This may be normal in the case of a front end Tier 1 web server inundated by user requests, where the application is run on other processors. However, heavy consumption of kernel mode may indicate an operational problem, possibly due to application malfunction, configuration or security.

It is useful in practice to know if a process consumes its entire time slice. Timeshared scheduling of processes is the most common scheduling algorithm for modern operating systems. In order to prevent scheduling lockout, operating systems with timeshared scheduling reduce the running priority of processes as they consume a time slice, while ready to run processes increase priority as they wait for a turn to run. In this way, many processor bound processes will sequence through execution, each getting a fair chance at a processor time slice. If the processes alternate in execution as displayed by a top command or process status list, the applications are likely consuming entire slices. Fair competition will tend to allocate even percentages for activity consuming time slices. Some operating systems (like linux) have ways of showing the consumption of time slices (in clock ticks).

A peculiar case of a thrashing processor occurs when a user visible process falls behind other active processes it uses to obtain its results. Even if each process doesn't use its full time slice, it still is interdependent on the other process. The time to reverse this priority inversion results in a bottleneck condition, where the CPU run queue grows, swaps off processes, and still doesn't't' get quite finished, resulting in processor thrash. Raising the priority of processes doesn't help, as they wait for the low priority condition to finish. Windows (and some UNIX derivatives like HPUX) operating systems are especially vulnerable to this condition, while Solaris shines in these cases.

In the gross sense, the server should contain enough memory to support server operations and applications without having to page or swap to disk. Finer grain analysis of the use of memory (see below) is a much more complicated subject, since memory isn't like a processor – it has indirect as well as direct effects. To locate direct server memory problems – those that impact the other hardware resources – we just need to know that there is enough memory to hold everything without causing the I/O and processor resource to compensate by shuffling memory to disk and thus reducing our hardware resource capacity with the abstract resource of virtual memory.

Abstract resources are those constructed by the operating system from a combination of processor, memory, and I/O. Typically a thread in the kernel, a common server process, a class of object, or a shared hardware device is the bottleneck as illustrated by the ps process status command. Many processes will be found to be blocked waiting for the abstract resource – if the abstract resource has an activity report, the capacity limit will be spotted. For example, almost all programs use files, so if file activity is high, the trick is to locate the process that is limiting performance.

Unlike hardware resources, it is often difficult to discern when abstract resources are at capacity, as hardware resource may be plentiful but fragmented. Abstract resource fragmentation occurs when the abstract resources need fragments of resource of other resources, both physical and abstract, to add an increment of resource to capacity. For example, when a program uses a file, it will need a file descriptor (or handle) for the program referral, a file instance for the kernel to maintain metadata, potentially disk block address structures or file allocation tables to refer to storage locations, underlying disk storage, bandwidth for data and metadata transfers, processor capacity to mediate the file operations that the program requires, and capacity for the I/O and processor to recover from recoverable faults, errors, and transient conditions that may occur during the operation.

Lets say that a server can support 10K/sec file record transfers using 128Mbytes of memory, 75% of the processor, and 80% of disk I/O rate capacity. If we only have 100 Mbytes and 70% of processor to give it, fragmentation of resource means that we'll be lucky to get something like 6-7 Kbytes/sec record transfers, and its doubtful that the disk I/O rate will approach even 60% in use. This interdependence on resources inherently limits capacity. In the management of processor and memory resources, OS design that minimizes resource fragmentation has the potential for 100x network communications improvement, and near doubling of sustained disk storage I/O transfer rate. 386BSD pioneered a novel page cache design and memory socket architecture to reap these gains.

OS Design Limits on Processor Performance

The processor allocates a bit of the processor, a time slice, to processes. It also gives them a weighted priority level in order to cycle through them. Typically, kernel processes have higher priority than user processes. As such, the operating system still is able to grab the processor as needed. The server is sluggish but running, and the operating system seems to function correctly. If the operating system has not been altered and the server is properly secured, it is reasonable to assume it isn't the problem.

Operating systems are primarily bookkeepers for resources. Part of being an excellent bookkeeper is careful resource utilization – the ability to hand out resources in such a way that the most desirable effect is achieved. For servers, this means awarding resources such that processes don't block. The more processes can run, the greater chance useful work may be done. In the time window of the server's operating system, the maximum achievable threshold on the run queue is frequently set by the number of system resources exclusively tied up by each process.

A historical accident of operating system design makes this difficult to directly observe. Many early operating systems were designed with the idea that the operating system kernel was simply the switchyard for access to services, with multiprogramming of tasks and processes invoked to implement the operating system's services. Services were delegated such that it would be possible to compartmentalize them, thus making the boundaries between services and the kernel more manageable. Unfortunately, expedient or opportunistic kernel designers found they produced a more immediate impact by dropping file system, networking, and other service code into the kernel, bypassing the more challenging goal of finishing the necessary interfaces to run them external to the system as utility user processes. This "BSD mistake" has been costly to many OSes that followed the trend, and results in limits to performance and resource monitoring.

Attempts to reverse this trend in operating systems (like MACH and HURD) have so far been unsuccessful. We believe the lack of success is due to too grand a vision that has been undercut by an impatient OS deployment base. Nevertheless, if the OS worked with a "pure kernel", it would be easier to notice the effectiveness of resource management. Now all of the major volume server OS implementations have "impure kernels" with hundreds of resources that create ad hoc resource commitments to support various services. These "spider webs" of resources overlap resource management of services with kernel resources to make minor gains on performance, yet wreck our ability to see distinct resource management within the kernel and its embedded services.

OS Design Limits on Memory Resource

Each process is allocated a piece of memory by the operating system for it to be processed. The allocation and deallocation of memory is primarily done independent of the processor resource, however. Scheduling of resource is a product of the memory allotted to a process and the time allotted to a process (the time slice). So memory resource problems are analogous to processor resource problems in terms of appearance and remedy.

Unlike the processor resource, it is difficult to distinguish between a single application consuming a great deal of memory resource and a gang of applications collectively consuming memory resource. Hence, memory reclamation is done in response to actual demands for memory use. The more memory that is required, the more memory will be reclaimed and assigned.

If an applications program requires a great deal more memory than is usually allocated, it lingers idle until the memory resource is found – risking the possibility of losing all of its already allocated resource before it grabs enough. If it loses its memory before it reaches the front of the CPU queue, it cannot be run.

Modern operating systems attempt to enforce fairness in memory resource allocation for general purpose use. While simple fairness schemes work for personal computers, programming, and the like, they often are insufficient for data center servers which must service a huge number of request, or a specialized server that must handle massive amounts of I/O and memory allocation, as does a database server or file server.

Memory Allocation in the Operating System

In order to understand how to use memory on a server, it helps to understand a little about how memory works in a server operating system. Typically, a server's operating system allocates gross amounts of memory only when it must, and pre-allocates small amounts of memory when it may subsume multiple allocations into a single one. As such, memory for abstract kernel resources is frequently over-allocated by the virtual memory system in the operating system assuming a strategy to pare back if we experience resource exhaustion through lazy evaluation.

Memory for a program's instruction, data, and stack is frequently allocated at the last possible moment, when it is first referenced. A technique used by the virtual memory system called copy on write allows memory to be shared until it is written, so that multiple processes containing the same program can share redundant memory as long as possible. That is in fact the whole point of the postponed allocation – to promote opportunities for more sharing.

Once an application is running, a memory heap is obtained from the operating system to subdivide the application's dynamic memory allocation. A Java or C++ Programmer, for example, would find that her object instances use the memory heap to implement the "new" instance's memory before invoking a constructor on a freshly minted object. Heap space often is very fragmented and it is difficult to return space from the application – usually, the server's operating system will age out unused part of the heap to the paging store, logically recovering the real memory for use as another piece of virtual memory. Object oriented programs in C++ and scripts in Python or Java sustain bursts of memory allocation as objects are instantiated at the start of the script and when the application's operations are performed – thus pre-allocation of memory is a real advantage for such applications.

I/O operations typically require pinned or non-paged memory. Operating systems that support deep write buffering may actually commit substantial amounts of memory in this state to allow very large and potentially double buffered write operations to sustain very high data rates. Pinned memory can't be reused, and may clog up the server's operating system until the memory becomes unpinned.

All these memory resources interact, but each element (physical, virtual, kernel memory) has different characteristics and produces different bottleneck problems. Some memory turns over rapidly, while other memory is allocated but seldom used. Some kinds of memory like pinned memory cannot be made virtual because it is used to pull things in and out (common in I/O and network connections – socket connection queues and protocol control blocks are pinned memory). Finally, if a networked server system handling tons of requests seems to appear memory starved but use profiles indicate plenty of memory, you probably need to check your socket queues.

OS Design Limits - Fracturing Memory and Server Performance

Memory capacity, unlike processor capacity, isn't usually seen in a single view like a processor run queue, but through examining a series of puzzles and interlinking their existence. There are many areas to examine, once the "flawless" system is the suspect. Software updates frequently expand their preallocation of memory for their own specific use. If the preallocation of memory has been miscalculated to consume the majority of memory (especially in the case of an operating system update which may have pinned far too much memory), there is much less available for everything else than prior to the update.

Errors in the new software installed may also have caused memory to be consumed and not recycled back for use again. This perpetual drain of memory can cause a server to gradually lose capacity over time. The processor overhead picks up because its scavenging for resources more frequently, and disk bandwidth increases as the size of its cache falls and its effectiveness at reducing disk I/O is reduced.

The greater the distance required for Internet communications, the greater the impact on memory management. Distant communications means greater time for packets to travel round trip between client and server. If the wrong packet is sent or missed, client and server will take a longer period of time to accommodate the error.

The Internet can handle multi-gigabit/sec transfer rates – a gigabit/sec(10^9 bits/sec) transfer recovering from an error on a common 100ms round trip latency will accumulate a backlog of 100 million bits or about the size of a common memory chip. When you combine great transfer capacities with large and varying delays, you get the potential for highly variable applications demand. And while you can throttle bandwidth outbound to moderate demand, you can't control the input of requests.

Over-allocation of memory means that other applications or processes can't use it. Frequently, applications programs have heap stores or dynamic memory allocation pools. One technique to improve performance is to allocate larger heaps to handle capacity issues through common applications. If you set them too large, then the number of applications that can be run is reduced.

Determining Limits - Application Resource Saturation Technique Using Fair Competition

Server status commands may truthfully report only a few percent of CPU, memory, disk and network resource used. Yet spikes in usage might in effect understate the impact of an application, leaving us misled as to the actual use of resources. We can't wait for OS vendors to improve their status commands – we've been using the same ones for close to 20 years now, and we don't think they're going to change much. So what can we do to get better information with the tools we've been given?

Server operating systems provide fair competition for resources. If any process asks for all resources, it will get as much as is available. If two processes ask for all resources, however, it isn't a win/lose competition, where one gets all and the other none. Instead, each process will get roughly half of what is available. The more processes requesting all resource, the more ways it is divided, so that three processes wanting all resource get 33%, four processes get 25%, and so forth.

As the next step, multiple copies of the consumption program are activated until the application performance is reduced. At this point, we know that the processor isn't shared equally by the number of hungry programs. From the load average (found by the la or w command) we can find the average number of processes N(proc). The reciprocal of this number 1/N(proc) is the fraction of the processor tied up by each of the processes.

OSPREY – UNIX Mindset in a Resource Rich World

UNIX was born on an incredibly resource starved machine (PDP 11). Its minimalist design was brilliant in reusing resources effectively, and every derivative afterward has industriously and microscopically followed this design goal. Our experiments with 386BSD showed that there were fundamental limits to this approach.

What if we view the OS issue not as minimizing low-level resource utilization, but instead the number of resource allocations across OS abstraction levels? This is the point of departure for OSPREY, which views fundamental OS components as built out of vertical resource allocation as a global capability list. If all of the resources can be allocated for a process, then a time slice can be scheduled, followed by a reclaim of freed resources.

Specialized hardware interfaces to networks, and entirely network driven I/O organization eliminate the need for all low level asynchronous processor/memory allocations, leaving the system able to do very high bandwidth network transfers directly to applications, such that the focus shifts to rates that can exceed many current bus bandwidth limits. The radical departure of OSPREY thus has in turn equally radical performance aspirations to match.

While the performance gains are immense, this implies fundamental design decisions from the outset. Incremental legacy migration impede the necessary design choice. Radical though such an approach might appear on first blush, the larger implications, especially regarding hardware ballistic processing mechanisms, could result in an entire new technology direction. Perhaps, given the dead-ends of so much of our current approaches, it might not be so radical after all.